Converting Grayscale Images to RGB Images Using Various Autoencoder Architectures

Christian Durant, Abhinav Kotta, Jackson Mazer

December 6, 2023

Abstract

In this project, we designed and experimented with various autoencoder architectures aimed toward converting grayscale images into their RGB counterparts. We utilized the 32x32 RGB images of the CIFAR10 dataset to test the models' conversion of their grayscale versions into their original RGB formats. The focus of the experimentation was to create a versatile model capable of generating RGB images from newly seen grayscale images and to identify the components of an autoencoder architecture that produce the best results. We tested a 3-layer convolutional model, a linear latent model, and a CNN latent model, and determined the 3-layer convolutional model produced the superior reconstructed colorized images.

1 Introduction

Throughout history, people have created brilliantly-colored architecture, clothing, paintings, and more. Any medium in which human creativity could be expressed witnessed an explosion of color. While we see their architecture as stark white in modern times, we know the ancient Romans bathed their statues, temples, and homes in splashes of bright colored paints, for example. Art movements throughout the centuries have always brought with them new trends and discoveries in color theory and creativity. Photography, then, became the next medium for popular colorful expression.

Color photography has been possible for over one hundred years, and has been widely found throughout the world for over fifty. Starting in the mid-1960s, the advent of widely-used color camera technology was brought to the business market, and in 1990, the first digital color camera was brought to the consumer market. Since then, camera technology has only improved to the point that the average person has a 12–48 megapixel camera in their pocket. But with such widely accessible high-quality color photography, the desire to colorize photos from before this time has become that much more popular. This desire is achievable with an autoencoder, and it is this desire that we sought to fulfill with this project.

2 Method

An autoencoder, or convolutional encoder-decoder model, is a type of layered convolutional neural network designed to compress and reconstruct an image. In doing so, the autoencoder seeks to predict the same image as an output as what it received as an input. As opposed to the typical one dataset input used with a convolutional neural network, we use two datasets when colorizing grayscale images. By presenting the encoder with colored images to learn the features from, and only then giving it grayscale images to reconstruct, the decoder will build back to the images using the color features learned initially. In this way, the autoencoder will break down a black-and-white image and build up a color image.

As a team, we decided to each develop a slightly different architecture for the encoder and decoder and each test them independently using the same dataset. Once that was completed, we would compare and contrast the results from each architecture and determine which was best for reconstruction.

3 Experiments

3.1 Dataset

The dataset we used for the autoencoder model was the CIFAR-10 dataset. This is a dataset composed of sixty-thousand 32x32 colored images separated into ten classes of five thousand training and one thousand test images each. It would make for a sufficient and easily-accessible dataset that isn't so big that training would take longer than necessary.

3.2 Evaluation metrics

For each of the models that we trained and employed, we used the mean-squared error loss function to evaluate the disparity between the autoencoders' output RGB images generated from their grayscale inputs and the original RGB images from which the grayscale images were derived. This method of evaluation allowed us to assess the autoencoders' abilities to generate the correct proportions and intensities of the three color channels of the outputs. We also took a more visual approach to evaluating the performance of the models by performing a direct comparison between the original RGB images and the output RGB images as shown below.

3.3 Results

One of the implementations of the autoencoder was an implementation using three convolutional layers as well as a dense layer in each of the encoder and decoder in Keras. The convolutional layers contain 64, 128, and 256 nodes respectively. This is a fairly standard implementation, and it produced quite good results with minimal loss and maximal accuracy.

Below are some images from the CIFAR-10 dataset next to the resultant colorized images produced by the autoencoder.

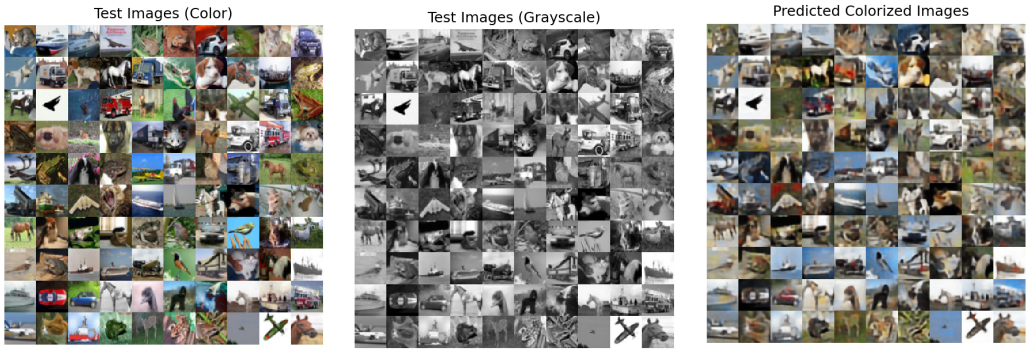

Figure 1: Images from the RGB test dataset, the grayscale test dataset, and the colorized results, respectively from the original model.

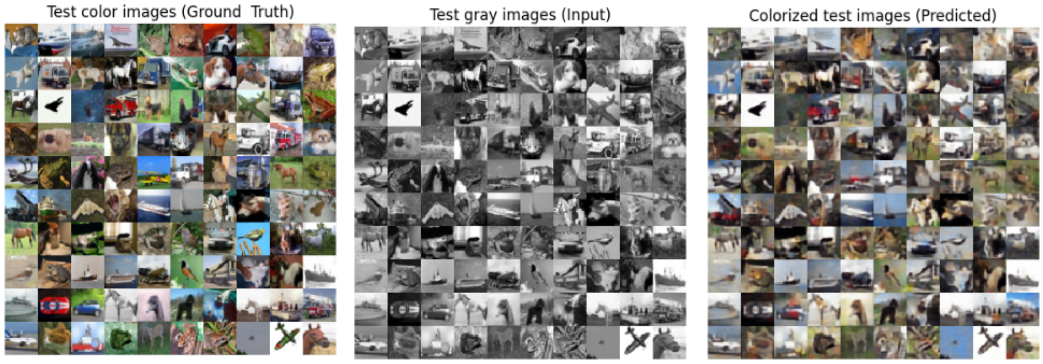

Figure 2: Output from the autoencoder architecture without a dynamically changing learning rate.

Further implementations of the autoencoder model attempted to identify the potential difference in performance between a fully connected (Dense) latent representation and a convolutional latent representation within the autoencoder architecture.

Two models were employed for such testing. One model had a fully-connected latent representation of a Linear layer consisting of 256 output nodes, while the other model had a fully convolutional latent representation consisting of 256 3x3 filters. For the fully connected latent representation, a second layer was required to upscale back to the number of neurons required for convolutional computation.

Below are the outputs of two autoencoder models, implemented in PyTorch.

Figure 3: Test input and colorized output from the autoencoder architecture containing a fully-connected latent representation.

Figure 4: Test input and colorized output from the autoencoder architecture containing a CNN representation.

3.4 Analysis and discussions

Upon examining the results of all of the models, it became more apparent what components of an autoencoder's architecture are most necessary to transform grayscale images into RGB format. The convolutional latent representation performed much better than the fully connected latent representation and also produced clearer images than that of the original configuration (Figure 1), as well. The original configuration, consisting of both convolutional layers and dense layers (similar to model 2), did manage to capture more color than the other two models, however.

The following analysis describes in detail each of the architectures that were employed throughout the experimentation as well as the change in the loss values during training. We can examine this information to determine each of the model's efficiency in learning to take the input grayscale images and generate their RGB counterparts.

3.4.1 Model Architecture and Training

```
Model: "encoder"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 encoder_input (InputLayer)  [(None, 32, 32, 1)]       0
 conv2d (Conv2D)             (None, 16, 16, 64)        640
 conv2d_1 (Conv2D)           (None, 8, 8, 128)         73856
 conv2d_2 (Conv2D)           (None, 4, 4, 256)         295168
 flatten (Flatten)           (None, 4096)              0
 latent_vector (Dense)       (None, 256)               1048832
=================================================================
Total params: 1418496 (5.41 MB)
Trainable params: 1418496 (5.41 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Model: "decoder"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 decoder_input (InputLayer)  [(None, 256)]             0
 dense (Dense)               (None, 4096)              1052672
 reshape (Reshape)           (None, 4, 4, 256)         0
 conv2d_transpose (Conv2DTr  (None, 8, 8, 256)         590080
 anspose)
 conv2d_transpose_1 (Conv2D  (None, 16, 16, 128)       295040
 Transpose)
 conv2d_transpose_2 (Conv2D  (None, 32, 32, 64)        73792
 Transpose)
 decoder_output (Conv2DTran  (None, 32, 32, 3)         1731
 spose)
=================================================================
Total params: 2013315 (7.68 MB)
Trainable params: 2013315 (7.68 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 5: First implementation of the autoencoder using three convolutional layers and a dense layer in both the encoder and the decoder.

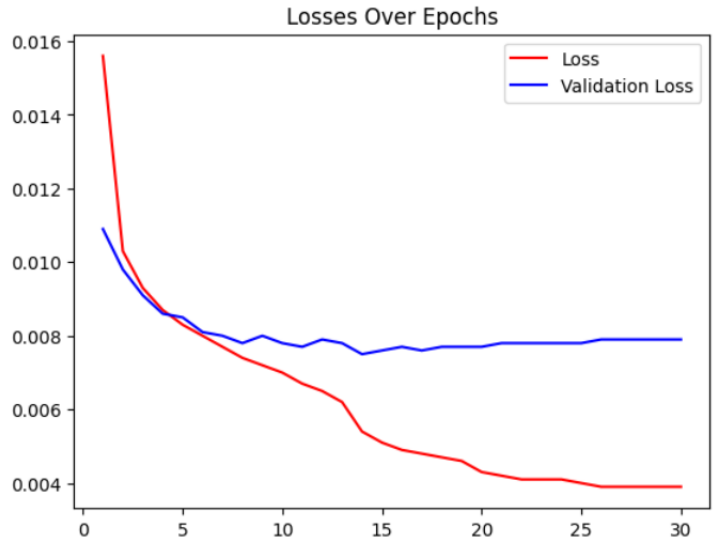

Figure 6: Loss values of the first model over the 30 epochs for which it was trained.

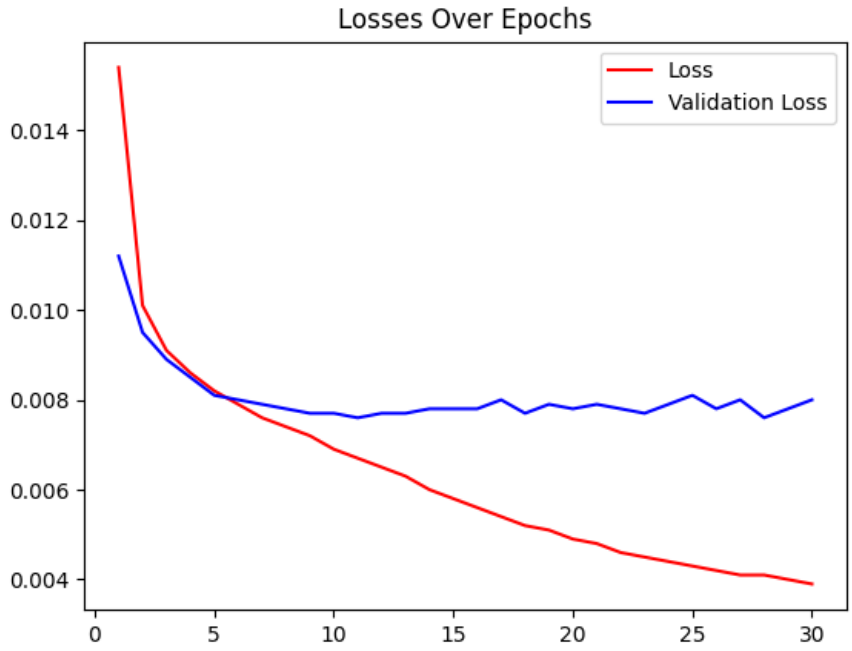

Figure 7: Loss values of the first model over the 30 epochs for which it was trained with a static learning rate.

```
==========================================================================================
├─Conv2d: 1-1                            [-1, 64, 32, 32]          640
├─Conv2d: 1-2                            [-1, 128, 32, 32]         73,856
├─MaxPool2d: 1-3                         [-1, 128, 16, 16]         --
├─Conv2d: 1-4                            [-1, 256, 16, 16]         295,168
├─MaxPool2d: 1-5                         [-1, 256, 8, 8]           --
├─Linear: 1-6                            [-1, 256]                 4,194,560
├─Linear: 1-7                            [-1, 16384]               4,210,688
├─Upsample: 1-8                          [-1, 256, 16, 16]         --
├─ConvTranspose2d: 1-9                   [-1, 128, 16, 16]         295,040
├─Upsample: 1-10                         [-1, 128, 32, 32]         --
├─ConvTranspose2d: 1-11                  [-1, 64, 32, 32]          73,792
├─ConvTranspose2d: 1-12                  [-1, 3, 32, 32]           1,731
==========================================================================================
Total params: 9,145,475
Trainable params: 9,145,475
Non-trainable params: 0
Total mult-adds (M): 312.74
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 2.90
Params size (MB): 34.89
Estimated Total Size (MB): 37.79
==========================================================================================
```

Figure 8: Architecture of the autoencoder model containing a Linear latent representation.

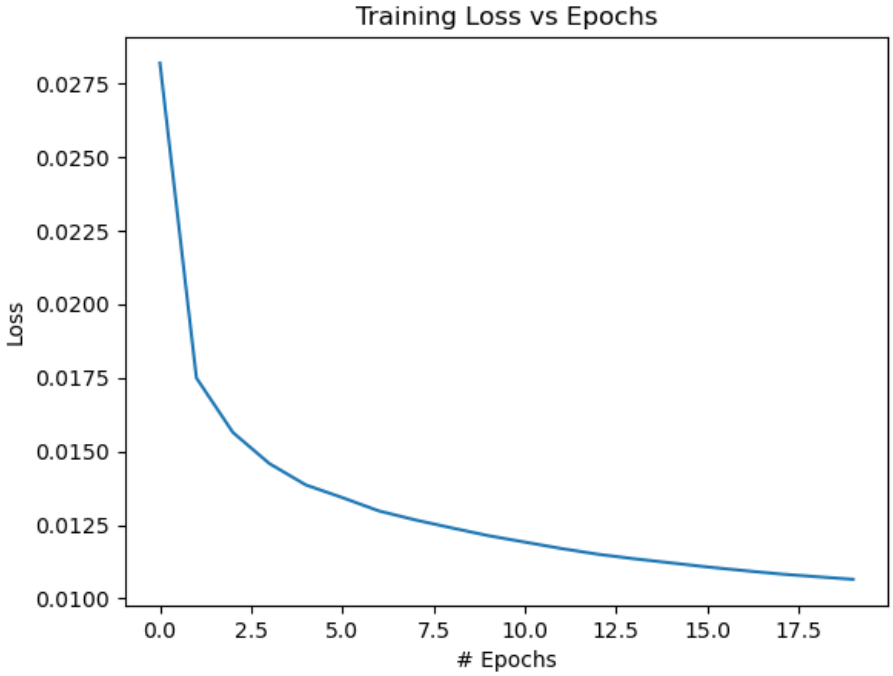

Figure 9: Loss values of the FC latent space model over the 20 epochs for which it was trained.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Conv2d: 1-1                            [-1, 64, 32, 32]          640
├─Conv2d: 1-2                            [-1, 128, 32, 32]         73,856
├─MaxPool2d: 1-3                         [-1, 128, 16, 16]         --
├─Conv2d: 1-4                            [-1, 256, 16, 16]         295,168
├─Conv2d: 1-5                            [-1, 256, 16, 16]         590,080
├─ConvTranspose2d: 1-6                   [-1, 128, 16, 16]         295,040
├─Upsample: 1-7                          [-1, 128, 32, 32]         --
├─ConvTranspose2d: 1-8                   [-1, 64, 32, 32]          73,792
├─ConvTranspose2d: 1-9                   [-1, 3, 32, 32]           1,731
==========================================================================================
Total params: 1,330,307
Trainable params: 1,330,307
Non-trainable params: 0
Total mult-adds (M): 455.34
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 3.27
Params size (MB): 5.07
Estimated Total Size (MB): 8.35
==========================================================================================
```

Figure 10: Architecture of the autoencoder model containing a convolutional latent representation.

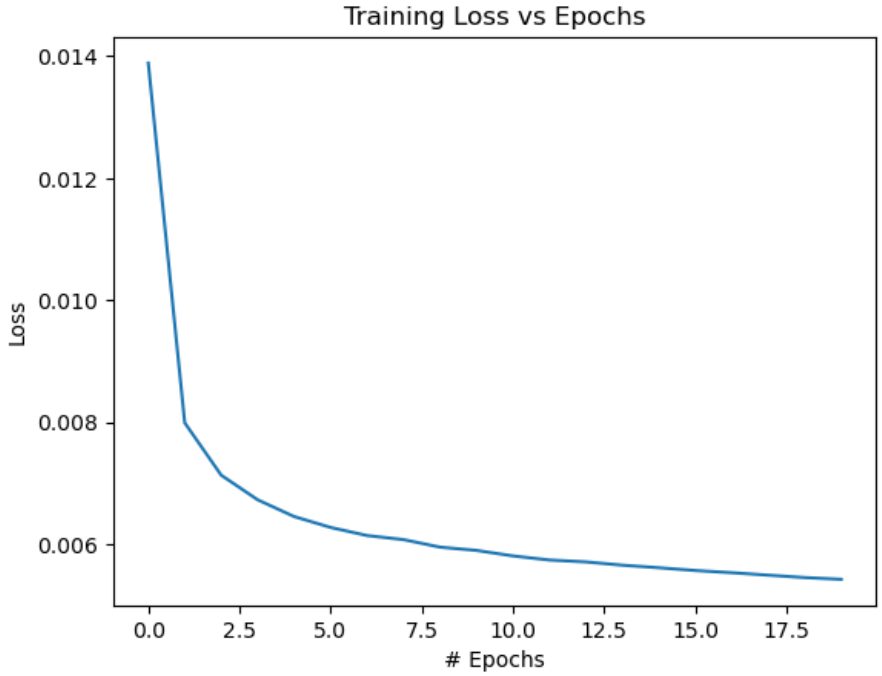

Figure 11: Loss values of the CNN latent space model over the 20 epochs for which it was trained.

As shown in Figures 5 and 6, the first architecture was capable of achieving a minimum loss on the training set of approximately 0.004, allowing it to generate the RGB images as shown in the results.

3.4.2 Comparisons

Figures 1-4 display the difference between having n fully connected and convolutional latent space representation within the autoencoder. The CNN architecture managed to achieve a minimum loss comparable to that of the first model at approximately 0.005, while the FC architecture struggled to even get below 0.01. The implications of this difference are demonstrated in the output, where the output of the model with the convolutional latent space is much more defined and clear than that of the model with the linear latent space. Additionally, the vibrancy of the images reconstructed using the model with the three convolutional layers is significantly more notable than their counterparts in the other architectures.

It is also worth noting that the model containing the fully-connected latent space had significantly more parameters than the other models that were employed, causing it to take more time to train during experimentation. The model with the convolutional latent space, on the other hand, had even fewer parameters than the first model but still achieved a comparable loss, suggesting that further modification to the architecture could yield similar or better results on the test set with a fully convolutional model.

Overall, it's clear that the 3-convolutional layer architecture produced the best-looking results. While its loss was comparable to that of the CNN latent space model, both at the 20 epoch mark and what could be extrapolated to be the loss beyond that mark for the latter, the apparent reconstruction was leagues ahead in the former. The performance of the linear latent representation architecture, then, is insufficient in both categories.

3.4.3 Discussion

The experimentation with 3-layer autoencoders, fully connected (Dense) latent, and convolutional representation of models yielded significant insights into its performance and adaptability in grayscale image colorization.

The observation that a static learning rate showed a slight degradation in the colorization quality underlines the importance of adaptive learning strategies. Dynamically adjusting the learning rate

when the loss becomes stationary for a certain number of epochs seems crucial for maintaining optimal convergence and enhancing the intricacies of colorized outputs, although makes a minimal difference when compared to other architectures.

The comparison between fully connected and convolutional latent spaces highlighted the superiority of convolutional architectures for capturing intricate spatial information. The convolutional latent space exhibited a more effective representation of image features by increasing the image quality compared to the fully connected latent representation.

Despite variations in architecture and latent space representation, the robustness and performance of the conventional 3-layer autoencoder stood out. Its popularity stems from its ability to strike a balance between model complexity and efficiency, offering a favorable trade-off for grayscale image colorization tasks. The adaptability and reliability of this architecture in consistently producing high-quality colorized outputs make it a preferred choice among the experimented configurations.

4 Conclusion

4.1 Description

In this work, we investigate the viability of multiple different autoencoder architectures for the colorization of grayscale images. We have tested a 3-layer model with and without modifications to its learning rate, and two latent space models, both linear and CNN-composed. While the fixed learning rate slightly impacted colorization quality, the comparative analysis underscored the effectiveness of the 3-layer autoencoder architecture. The superior performance of the convolutional latent space reaffirmed its significance in preserving spatial information crucial for accurate colorization. Future research focusing on hybrid architectures integrating dynamic learning rates with convolutional latent spaces holds promise for further enhancing colorization fidelity. These insights serve as valuable guidelines for optimizing autoencoder architectures for grayscale image colorization tasks, paving the way for more refined and realistic colorization techniques in the future.

4.2 What I Learned

Through the implementation of two separate models for the task of converting grayscale images to RGB images, I learned about the impact that various architectures and hyperparameters may have on the effectiveness of a neural network. The experimentation of the different architectures and hyperparameters was extensive, and employing efficient methods towards changing architectures between training is very important for such research.

I also learned that convolutional networks are far more powerful for processing visual information compared to similar models utilizing linear layers. When comparing the number of parameters and accuracy between models employing CNNs and FC layers, the models with CNN layers are far more efficient for certain tasks and are more robust.

5 Contribution

5.1 Code

Below is the code for the different architectures examined in this work.

5.1.1 Three Convolutional Layer Implementation With Learning Rate Modification

```
import numpy as np
import matplotlib.pyplot as plt
import os
from keras.layers import Dense, Input
from keras.layers import Conv2D, Flatten
from keras.layers import Reshape, Conv2DTranspose
from keras.models import Model
from keras.callbacks import ReduceLROnPlateau
from keras.callbacks import ModelCheckpoint
from keras.datasets import cifar10
```

```
from keras import backend as K

'---------------'
'PREPARING THE DATASETS'
'---------------'
# Load in the images from the cifar10 dataset.
(x_train, _), (x_test, _) = cifar10.load_data()

# Turn the images in the test dataset to grayscale for a new test dataset
x_train_gray = np.dot(x_train[...,:3], [0.299, 0.587, 0.114])
x_test_gray = np.dot(x_test[...,:3], [0.299, 0.587, 0.114])

# Get the size of the images
img_rows = x_train.shape[1]
img_cols = x_train.shape[2]
channels = x_train.shape[3]

'---------------'
'PLOTTING THE DATASETS'
'---------------'
# Store 100 images of the color test dataset
datasetC = x_test[:100]
datasetC = datasetC.reshape((10, 10, img_rows, img_cols, channels))
datasetC = np.vstack([np.hstack(i) for i in datasetC])

# Plot 100 images of the color test dataset
plt.figure()
plt.axis('off')
plt.title('Test Images (Color)')
plt.imshow(datasetC, interpolation='none')
plt.show()

# Store 100 images of the grayscale test dataset
datasetG = x_test_gray[:100]
datasetG = datasetG.reshape((10, 10, img_rows, img_cols))
datasetG = np.vstack([np.hstack(i) for i in datasetG])

# Plot 100 images of the grayscale test dataset
plt.figure()
plt.axis('off')
plt.title('Test Images (Grayscale)')
plt.imshow(datasetG, interpolation='none', cmap='gray')
plt.show()

'---------------'
'PREPARING FOR MODEL INPUT/OUTPUT'
'---------------'
# Normalize and reshape the color datasets
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, channels)
x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, channels)

# Normalize and reshape the grayscale datasets
x_train_gray = x_train_gray.astype('float32') / 255
x_test_gray = x_test_gray.astype('float32') / 255
x_train_gray = x_train_gray.reshape(x_train_gray.shape[0], img_rows, img_cols, 1)
x_test_gray = x_test_gray.reshape(x_test_gray.shape[0], img_rows, img_cols, 1)

# Define parameters
input_shape = (img_rows, img_cols, 1)
batch_size = 32
kernel_size = 3
latent_dim = 256
layer_filters = [64, 128, 256]

'---------------'
'BUILDING THE ENCODER MODEL'
'---------------'

# Build the model
```

```
inputs = Input(shape=input_shape, name='encoder_input')
x = inputs

# 3 Convolution layers: 64 filters, 128 filters, 256 filters
for filters in layer_filters:
    x = Conv2D(filters=filters,
               kernel_size=kernel_size,
               strides=2,
               activation='relu',
               padding='same')(x)

# Reshape to 4x4x256 for processing, to eventually be shaped back into 32x32x3
shape = K.int_shape(x)

# Thank you for resources online on how to do this type of thing.
x = Flatten()(x)
latent = Dense(latent_dim, name='latent_vector')(x)

encoder = Model(inputs, latent, name='encoder')
encoder.summary()

'---------------'
'BUILDING THE DECODER MODEL'
'---------------'
latent_inputs = Input(shape=(latent_dim,), name='decoder_input')
x = Dense(shape[1]*shape[2]*shape[3])(latent_inputs)
x = Reshape((shape[1], shape[2], shape[3]))(x)

# Three convolution layers: 256 filters, 128 filters, 64 filters
for filters in layer_filters[::-1]:
    x = Conv2DTranspose(filters=filters,
                        kernel_size=kernel_size,
                        strides=2,
                        activation='relu',
                        padding='same')(x)

outputs = Conv2DTranspose(filters=channels,
                          kernel_size=kernel_size,
                          activation='sigmoid',
                          padding='same',
                          name='decoder_output')(x)

decoder = Model(latent_inputs, outputs, name='decoder')
decoder.summary()

'---------------'
'THE AUTOENCODER'
'---------------'
autoencoder = Model(inputs, decoder(encoder(inputs)), name='autoencoder')
autoencoder.summary()

# Save as a new model any time the loss improves.
save_dir = os.path.join(os.getcwd(), 'saved_models')
model_name = 'Model{epoch:02d}.h5'
if not os.path.isdir(save_dir):
        os.makedirs(save_dir)
filepath = os.path.join(save_dir, model_name)

# Reduce learning rate by sqrt(0.1) if the loss does not improve in 5 epochs
lr_reducer = ReduceLROnPlateau(factor=np.sqrt(0.1),
                               cooldown=0,
                               patience=5,
                               verbose=1,
                               min_lr=0.5e-6)

# Save weights
checkpoint = ModelCheckpoint(filepath=filepath,
                             monitor='val_loss',
                             verbose=1,
                             save_best_only=True)

```

```
# Mean Square Error (MSE) loss function, Adam optimizer
autoencoder.compile(loss='mse', optimizer='adam')

# Called every epoch
callbacks = [lr_reducer, checkpoint]

# Train the autoencoder
autoencoder.fit(x_train_gray,
                x_train,
                validation_data=(x_test_gray, x_test),
                epochs=30,
                batch_size=batch_size,
                callbacks=callbacks)

# Predict the autoencoder output from test data
x_decoded = autoencoder.predict(x_test_gray)

# Display the 1st 100 colorized images
datasetGtoC = x_decoded[:100]
datasetGtoC = datasetGtoC.reshape((10, 10, img_rows, img_cols, channels))
datasetGtoC = np.vstack([np.hstack(i) for i in datasetGtoC])
plt.figure()
plt.axis('off')
plt.title('Predicted Colorized Images')
plt.imshow(datasetGtoC, interpolation='none')
plt.show()
```

5.1.2 Three Convolutional Layer Implementation Without Learning Rate Modification

```
'''Colorization autoencoder

Used to train gray scale images of CIFAR-10 dataset to make them colorized
'''

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.layers import Conv2D, Flatten
from tensorflow.keras.layers import Reshape, Conv2DTranspose
from tensorflow.keras.models import Model
from tensorflow.python.keras.callbacks import ReduceLROnPlateau
from tensorflow.python.keras.callbacks import ModelCheckpoint
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import plot_model
from tensorflow.keras import backend as K

import numpy as np
import matplotlib.pyplot as plt
import os

def rgb2gray(rgb):
    """Convert from color image (RGB) to grayscale.
    """
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])


# load the CIFAR10 data
(x_train, _), (x_test, _) = cifar10.load_data()

# input image dimensions
# we assume data format "channels_last"
img_rows = x_train.shape[1]
img_cols = x_train.shape[2]
channels = x_train.shape[3]

# create saved_images folder
imgs_dir = 'saved_images'
save_dir = os.path.join(os.getcwd(), imgs_dir)
if not os.path.isdir(save_dir):
```

```
43         os.makedirs(save_dir)
44
45 # display the 1st 100 input images (color and gray)
46 imgs = x_test[:100]
47 imgs = imgs.reshape((10, 10, img_rows, img_cols, channels))
48 imgs = np.vstack([np.hstack(i) for i in imgs])
49 plt.figure()
50 plt.axis('off')
51 plt.title('Test color images (Ground  Truth)')
52 plt.imshow(imgs, interpolation='none')
53 plt.savefig('%s/test_color.png' % imgs_dir)
54 plt.show()
55
56 # convert color train and test images to gray
57 x_train_gray = rgb2gray(x_train)
58 x_test_gray = rgb2gray(x_test)
59
60 # display grayscale version of test images
61 imgs = x_test_gray[:100]
62 imgs = imgs.reshape((10, 10, img_rows, img_cols))
63 imgs = np.vstack([np.hstack(i) for i in imgs])
64 plt.figure()
65 plt.axis('off')
66 plt.title('Test gray images (Input)')
67 plt.imshow(imgs, interpolation='none', cmap='gray')
68 plt.savefig('%s/test_gray.png' % imgs_dir)
69 plt.show()
70
71
72 # normalize output train and test color images
73 x_train = x_train.astype('float32') / 255
74 x_test = x_test.astype('float32') / 255
75
76 # normalize input train and test grayscale images
77 x_train_gray = x_train_gray.astype('float32') / 255
78 x_test_gray = x_test_gray.astype('float32') / 255
79
80 # reshape images to row x col x channel for CNN output/validation
81 x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, channels)
82 x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, channels)
83
84 # reshape images to row x col x channel for CNN input
85 x_train_gray = x_train_gray.reshape(x_train_gray.shape[0], img_rows, img_cols, 1)
86 x_test_gray = x_test_gray.reshape(x_test_gray.shape[0], img_rows, img_cols, 1)
87
88 # network parameters
89 input_shape = (img_rows, img_cols, 1)
90 batch_size = 32
91 kernel_size = 3
92 latent_dim = 256
93 # encoder/decoder number of CNN layers and filters per layer
94 layer_filters = [64, 128, 256]
95
96 # build the autoencoder model
97 # first build the encoder model
98 inputs = Input(shape=input_shape, name='encoder_input')
99 x = inputs
100 # stack of Conv2D(64)-Conv2D(128)-Conv2D(256)
101 for filters in layer_filters:
102     x = Conv2D(filters=filters,
103                kernel_size=kernel_size,
104                strides=2,
105                activation='relu',
106                padding='same')(x)
107
108 # shape info needed to build decoder model so we don't do hand computation
109 # the input to the decoder's first Conv2DTranspose will have this shape
110 # shape is (4, 4, 256) which is processed by the decoder back to (32, 32, 3)
111 shape = K.int_shape(x)
112
113 # generate a latent vector
```

```
x = Flatten()(x)
latent = Dense(latent_dim, name='latent_vector')(x)

# instantiate encoder model
encoder = Model(inputs, latent, name='encoder')
encoder.summary()

# build the decoder model
latent_inputs = Input(shape=(latent_dim,), name='decoder_input')
x = Dense(shape[1]*shape[2]*shape[3])(latent_inputs)
x = Reshape((shape[1], shape[2], shape[3]))(x)

# stack of Conv2DTranspose(256)-Conv2DTranspose(128)-Conv2DTranspose(64)
for filters in layer_filters[::-1]:
    x = Conv2DTranspose(filters=filters,
                        kernel_size=kernel_size,
                        strides=2,
                        activation='relu',
                        padding='same')(x)

outputs = Conv2DTranspose(filters=channels,
                          kernel_size=kernel_size,
                          activation='sigmoid',
                          padding='same',
                          name='decoder_output')(x)

# instantiate decoder model
decoder = Model(latent_inputs, outputs, name='decoder')
decoder.summary()

# autoencoder = encoder + decoder
# instantiate autoencoder model
autoencoder = Model(inputs, decoder(encoder(inputs)), name='autoencoder')
autoencoder.summary()

# prepare model saving directory.
save_dir = os.path.join(os.getcwd(), 'saved_models')
model_name = 'colorized_ae_model.{epoch:03d}.h5'
if not os.path.isdir(save_dir):
        os.makedirs(save_dir)
filepath = os.path.join(save_dir, model_name)

# save weights for future use (e.g. reload parameters w/o training)
checkpoint = ModelCheckpoint(filepath=filepath,
                             monitor='val_loss',
                             verbose=1,
                             save_best_only=True)

# Mean Square Error (MSE) loss function, Adam optimizer
autoencoder.compile(loss='mse', optimizer='adam')

# called every epoch
callbacks = [checkpoint]

# train the autoencoder
autoencoder.fit(x_train_gray,
                x_train,
                validation_data=(x_test_gray, x_test),
                epochs=30,
                batch_size=batch_size,
                callbacks=callbacks)

# predict the autoencoder output from test data
x_decoded = autoencoder.predict(x_test_gray)

# display the 1st 100 colorized images
imgs = x_decoded[:100]
imgs = imgs.reshape((10, 10, img_rows, img_cols, channels))
imgs = np.vstack([np.hstack(i) for i in imgs])
plt.figure()
plt.axis('off')
```

```
plt.title('Colorized test images (Predicted)')
plt.imshow(imgs, interpolation='none')
plt.savefig('%s/colorized.png' % imgs_dir)
plt.show()
```

5.1.3 Linear and Convolutional Latent Space Implementations

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import time
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
import numpy as np
import random
from torchsummary import summary


# In[2]:


# Information about device
print(torch.cuda.device_count())
print(torch.cuda.current_device())
print(torch.cuda.device(0))
print(torch.cuda.get_device_name(0))

use_cuda = torch.cuda.is_available()
print(use_cuda)
# Set proper device based on cuda availability
device = torch.device("cuda" if use_cuda else "cpu")
print("Torch device selected: ", device)


# In[3]:


# Construct Autoencoder
class Net1(nn.Module):
    def __init__(self):
        super(Net1, self).__init__()

        # Define layers of the autoencoder neural network

        # Encoder
        self.conv1 = nn.Conv2d(1, 64, 3, padding=1)
        self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
        self.maxpool1 = nn.MaxPool2d(2)
        self.conv3 = nn.Conv2d(128, 256, 3, padding=1)
        self.maxpool2 = nn.MaxPool2d(2)

        # Code (Latent Representation)
        self.fc1 = nn.Linear(256*8*8, 256)
        self.fc2 = nn.Linear(256, 256*8*8)

        # Decoder
        self.upsample1 = nn.Upsample(scale_factor=2, mode='bilinear')
        self.deconv1 = nn.ConvTranspose2d(256, 128, 3, padding=1)
        self.upsample2 = nn.Upsample(scale_factor=2, mode='bilinear')
        self.deconv2 = nn.ConvTranspose2d(128, 64, 3, padding=1)
        self.deconv3 = nn.ConvTranspose2d(64, 3, 3, padding=1)

    def forward(self, X):

        # Encoder
        output = F.relu(self.conv1(X))
```

```
        output = F.relu(self.maxpool1(self.conv2(output)))
        output = F.relu(self.maxpool2(self.conv3(output)))

        # Latent Representation
        output = torch.flatten(output,1) # Flatten
        output = F.relu(self.fc1(output))
        output = F.relu(self.fc2(output))
        output = torch.reshape(output, (-1, 256, 8, 8)) # Reshape

        # Decoder
        output = F.relu(self.deconv1(self.upsample1(output)))
        output = F.relu(self.deconv2(self.upsample2(output)))
        output = F.sigmoid(self.deconv3(output))

        return output

# Construct Autoencoder
class Net2(nn.Module):
    def __init__(self):
        super(Net2, self).__init__()

        # Define layers of the autoencoder neural network

        # Encoder
        self.conv1 = nn.Conv2d(1, 64, 3, padding=1)
        self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
        self.maxpool1 = nn.MaxPool2d(2)
        self.conv3 = nn.Conv2d(128, 256, 3, padding=1)

        # Code (Latent Representation)
        self.conv4 = nn.Conv2d(256, 256, 3, padding=1)

        # Decoder
        self.deconv1 = nn.ConvTranspose2d(256, 128, 3, padding=1)
        self.upsample1 = nn.Upsample(scale_factor=2, mode='bilinear')
        self.deconv2 = nn.ConvTranspose2d(128, 64, 3, padding=1)
        self.deconv3 = nn.ConvTranspose2d(64, 3, 3, padding=1)

    def forward(self, X):

        # Encoder
        output = F.relu(self.conv1(X))
        output = F.relu(self.maxpool1(self.conv2(output)))
        output = F.relu(self.conv3(output))

        # Latent Representation
        output = self.conv4(output)

        # Decoder
        output = F.relu(self.deconv1(output))
        output = F.relu(self.deconv2(self.upsample1(output)))
        output = F.sigmoid(self.deconv3(output))

        return output


# In[4]:


import torchvision
import torchvision.transforms as transforms
import torch.optim as optim
from torchvision.transforms.functional import rgb_to_grayscale


# In[5]:


# Transforms
train_transform = transforms.Compose(
    [transforms.ToTensor(),
```

```
136      transforms.RandomRotation(15)]
137     )
138
139 test_transform = transforms.Compose(
140     [transforms.ToTensor()]
141     )
142
143 # Load in CIFAR10 data, splitting training and test data and applying transforms
144 train_data = torchvision.datasets.CIFAR10("./", train= True, transform =
        train_transform, download = True)
145 test_data = torchvision.datasets.CIFAR10("./", train= False, transform =
        test_transform, download = True)
146
147 # Create dataloaders for loading in data in batches of size mini_batch_size
148 mini_batch_size = 100
149 train_loader = torch.utils.data.DataLoader(train_data, batch_size=mini_batch_size,
        shuffle=True, num_workers=2)
150 test_loader = torch.utils.data.DataLoader(test_data, batch_size=mini_batch_size,
        shuffle=False, num_workers=2)
151
152
153 # In[6]:
154
155
156 # Define the training function
157 def train(model, num_epochs, optimizer, loss_func):
158     # Set model to training mode
159     model.train()
160     losses = []
161
162     for i in range(num_epochs):
163         total_loss = 0
164
165         for (images, _) in train_loader:
166             images = images.to(device) # Push tensors to GPU
167             grayscale_images = rgb_to_grayscale(images) # Convert images to grayscale
168             optimizer.zero_grad() # Zero the gradients
169             outputs = model(grayscale_images) # Forward pass
170             loss = loss_func(outputs, images) # Calculate loss
171             loss.backward() # Backpropagation
172             optimizer.step() # Update weights
173             total_loss += loss.item() # Accumulate loss
174
175         # Save loss for graphing
176         losses.append(total_loss / len(train_loader))
177         print(f'(Epoch {i+1}) Training Loss: {total_loss / len(train_loader)} Test
        Loss: {test(model)}')
178
179     return losses
180
181 # Define function for applying model to test data and returning the accuracy
182 def test(model):
183     # Set model to evaluation mode
184     model.eval()
185     total_loss = 0
186
187     # Test on test set
188     for (images, _) in test_loader:
189         images = images.to(device) # Push tensors to GPU
190         grayscale_images = rgb_to_grayscale(images) # Convert images to grayscale
191         outputs = model(grayscale_images) # Forward pass
192         loss = F.mse_loss(outputs, images) # Calculate loss
193         total_loss += loss.item() # Accumulate loss
194
195     return total_loss/len(test_loader)
196
197
198 # In[7]:
199
200
201 # Define hyperparameters
```

```
learning_rate = 1e-4
num_epochs = 20


# In[10]:


# Create the model with FC latent code
net1 = Net1().to(device)

# Print a summary of the model WIP
summary(net1, (1, 32, 32))

# Define the loss function and optimizer
loss_func = nn.MSELoss()
optimizer = optim.Adam(net1.parameters(), lr=learning_rate)

# Load model, if saved parameters exist
try:
    net1.load_state_dict(torch.load("./saved models/net1.pth"))
except:
    # Perform training
    training_losses = train(net1, num_epochs, optimizer, loss_func)

    # Save model
    torch.save(net1.state_dict(), './saved models/net1.pth')

# Prepare model for evaluation
net1.eval()


# In[11]:


# Plot the losses over epochs for model 1
plt.plot(training_losses)
plt.title("Training Loss vs Epochs")
plt.xlabel("# Epochs")
plt.ylabel("Loss")
plt.show()


# In[12]:


# Create the model with CNN latent code
net2 = Net2().to(device)

# Print a summary of the model WIP
summary(net2, (1, 32, 32))

# Define the loss function and optimizer
loss_func = nn.MSELoss()
optimizer = optim.Adam(net2.parameters(), lr=learning_rate)

# Load model, if saved parameters exist
try:
    net2.load_state_dict(torch.load("./saved models/net2.pth"))
except:
    # Perform training
    training_losses = train(net2, num_epochs, optimizer, loss_func)

    # Save model
    torch.save(net2.state_dict(), './saved models/net2.pth')

# Prepare model for evaluation
net2.eval()


# In[13]:

```

```
273
274 # Plot the losses over epochs for model 2
275 plt.plot(training_losses)
276 plt.title("Training Loss vs Epochs")
277 plt.xlabel("# Epochs")
278 plt.ylabel("Loss")
279 plt.show()
280
281
282 # In[ ]:
283
284
285 # Grab images for testing
286 test_images, _ = next(iter(test_loader))
287
288
289 # In[ ]:
290
291
292 ### Test Net1 ###
293
294 # Display images to be tested
295 display_test = np.array([transforms.ToPILImage()(img) for img in test_images])
296 display_test = display_test.reshape((10, 10, 32, 32, 3))
297 display_test = np.vstack([np.hstack(i) for i in display_test])
298 plt.figure()
299 plt.axis('off')
300 plt.title('Test Images (RGB)')
301 plt.imshow(display_test, interpolation='none')
302 plt.show()
303
304 # Convert RGB test images into grayscale
305 grayscale_images = rgb_to_grayscale(test_images).to(device)
306
307 # Display images in grayscale
308 display_test = np.array([transforms.ToPILImage()(img) for img in grayscale_images])
309 display_test = display_test.reshape((10, 10, 32, 32, 1))
310 display_test = np.vstack([np.hstack(i) for i in display_test])
311 plt.figure()
312 plt.axis('off')
313 plt.title('Test Images (Grayscale)')
314 plt.imshow(display_test, interpolation='none', cmap='gray')
315 plt.show()
316
317 # Run model on grayscale images
318 output = net1(grayscale_images)
319
320 # Display images after decoding
321 display_test = np.array([transforms.ToPILImage()(img) for img in output])
322 display_test = display_test.reshape((10, 10, 32, 32, 3))
323 display_test = np.vstack([np.hstack(i) for i in display_test])
324 plt.figure()
325 plt.axis('off')
326 plt.title('Test Images (FC Code Autoencoder)')
327 plt.imshow(display_test, interpolation='none')
328 plt.show()
329
330
331 # In[ ]:
332
333
334 ### Test Net2 ###
335
336 # Display images to be tested
337 display_test = np.array([transforms.ToPILImage()(img) for img in test_images])
338 display_test = display_test.reshape((10, 10, 32, 32, 3))
339 display_test = np.vstack([np.hstack(i) for i in display_test])
340 plt.figure()
341 plt.axis('off')
342 plt.title('Test Images (RGB)')
343 plt.imshow(display_test, interpolation='none')
```

```
plt.show()

# Convert RGB test images into grayscale
grayscale_images = rgb_to_grayscale(test_images).to(device)

# Display images in grayscale
display_test = np.array([transforms.ToPILImage()(img) for img in grayscale_images])
display_test = display_test.reshape((10, 10, 32, 32, 1))
display_test = np.vstack([np.hstack(i) for i in display_test])
plt.figure()
plt.axis('off')
plt.title('Test Images (Grayscale)')
plt.imshow(display_test, interpolation='none', cmap='gray')
plt.show()

# Run model on grayscale images
output = net2(grayscale_images)

# Display images after decoding
display_test = np.array([transforms.ToPILImage()(img) for img in output])
display_test = display_test.reshape((10, 10, 32, 32, 3))
display_test = np.vstack([np.hstack(i) for i in display_test])
plt.figure()
plt.axis('off')
plt.title('Test Images (Fully CNN Autoencoder)')
plt.imshow(display_test, interpolation='none')
plt.show()
```

5.2 My Contribution

My contribution was primarily the typical three convolutional layer autoencoder implementation. I researched various possible implementations for autoencoders, knowing that's the direction we were going to take as a team, and noticed that many examples utilized three convolutional layers. As such, I deemed it sufficient as something of a baseline implementation. Following the model of various implementations discovered through my research, I included a novel way to nudge the loss along and improve the efficiency of my code by reducing the learning rate slightly if the loss plateaus.

In addition to the standard three convolutional layer autoencoder, I was also the one responsible for sections 1, 2, and 3.1, the Introduction, Method, and Dataset sections respectively. Additionally, I contributed to sections 3.3, 3.4, 4, and 5.1–Results, Analysis and Discussions, Conclusion, and Code–just as the others did. Specifically, I contributed the code for 5.1.1, Three Convolutional Layer Implementation With Learning Rate Modification.